A Benchmark for Multicore Machines

Frédéric Boussinot

A Benchmark for Multicore Machines (Note)

Frédéric Boussinot*
EMP-CMA/INRIA - Sophia Antipolis
B.P. 93, 06902 Sophia-Antipolis Cedex, France
Frederic.Boussinot@sophia.inria.fr

September 2007

Abstract

Simulation of collision of particles is proposed as a benchmark program for multicore machines. The focus is put on synchronisation and communication of threads. Some results obtained on a dual-core machine are presented.

1 Introduction

It is not so easy to imagine benchmarks showing the benefit of multicore machines. Of course, there is no difficulty to simultaneously use the processors by launching together several applications (or several times the same application). This is not what we are looking for, which can be formulated as: how can *a single application* benefit from a multicore architecture? The standard answer to this question is *multithreading*, which means that the application is decomposed in several threads that can be executed in real parallelism by the processors.

An example of multithreaded applications are Web servers (e.g. Apache servers) in which requests are implemented as threads. However, in servers, threads basically do not communicate and are quite autonomous and independent computing entities. Actually, servers do not really exploit the shared memory which is at the basis of multicore architectures. Servers are, thus, only partial benchmarks for multicore machines.

In the general context of multithreading, the issue of concurrent accesses to the memory shared by threads is immediately raised. Concurrent accesses to the same memory location possibly produce so-called *data-races* that can lead to data corruption and unpredictable behaviors. Thus, the question should be reformulated as follows: how can a single application, coded by a set of

*with support from ACI ALIDECS

communicating and synchronising threads, *safely* (without data-races) benefit of a multicore architecture?

We propose to consider the graphical simulation of a set of colliding particles as a benchmark for multicore machines. Collision processing is basically an algorithm whose complexity is square in the number of particles (actually $n^2/2$, where n is the number of particles). Thus, the amount needed of computing ressource can grow very rapidly as the number of particles increases. Each particle can be involved in several concurrent steps of collision processing: there is thus a need for protecting the data associated to particles. Moreover, the parallel threads should periodically synchronise, in order to get a realistic simulation (otherwise, a subset of particles could stay idle, while another subset is animated several times).

2 Standard Solutions

Let us suppose for simplicity that there are only two cores. In a standard approach, one should have two threads processing particles, each thread being possibly mapped to a core. One should distinguish two sets of particles: the particles to be processed, and the ones already processed. When all particles are processed, the two sets are exchanged and a new cycle begins.

Particles should be protected from concurrent accesses. This can be done by associating a lock to each particle.

Note that the issue can be rather complex when a first particle is being processed by a thread, while the other thread is processing a second particle and tries to determine if a collision can occur with the first particle. Then, a deadlock could occur if the particle lock is taken during all the time the particle is processed.

Another approach would be to divide particles in two sets, each one being processed by a specific thread. The problem then would be to let a thread access information concerning a particle managed by the other thread. Two variants exist for this approach: in the first one, division in two sets is made once for all; in the second variant, particles dynamically move from one set to the other. In the second variant, belonging to a set can be implemented to reflect *geometric proximity*. In this case, collisions can be processed totally independently by the two threads, except at the border of the two sets. This approach gives a way to break the complexity of collision processing. However, the issue of particles at the border of the two sets remains to be handled. This means that both variants have to consider the issue for a thread of accessing information processed by the other thread.

Note that a static (as opposed to dynamic) distribution of particles gives a simple solution to the problem of load balancing for the two executing threads, even if particles are not equally geometrically distributed in the simulation.

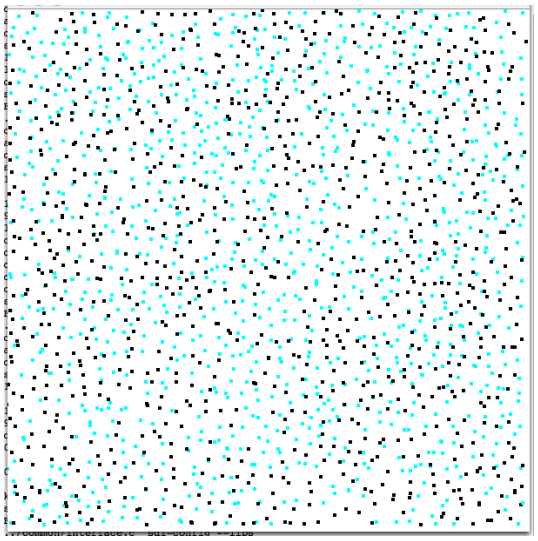

Figure 1: Simulation Snapshot

3 Benchmark Proposal

One considers a simulation made of particles having an inertial movement, and bouncing on the borders of the window. Moreover, collision of particles is processed by using a simple brute force algorithm. The particles are statically partitioned in two sets of equal size. Each set is animated by a thread and the challenge is to execute these two threads in the most parallel way, on the two cores of a dual-core machine. The implementation should be checked against data-races (ideally, absence of data-races should be proved). The number of simulated particles should typically be of several thousands.

The benchmark has been coded in FunLoft[1], a recently proposed concurrent language (the code is given in annex), with results described below.

The machine characteristics are: a Mac running Mac OS X 10.4.10, processor Intel Core 2 Duo, 2.33 GHz, 2GB of memory. Graphics is based on SDL[4].

The simulation with 2000 particles is shown in Figure 1 (1000 particles in one color are animated by one thread and 1000 particles in another color are animated by the other thread).

The CPU usage is shown in Figure 2 (graphics window masked during the measure). This is to compare with the usage obtained with the single threaded version of the benchmark, shown in Figure 3.

The time for simulating 100 instants and 1000 particles (one instant corresponds to the execution of all the 1000 particles) is shown Figure 4, with a memory footprint of about 30MB. Note that the total number of performed interactions is about $100 * 1000^2 = 10^8$.

For 2000 particles, the results are shown in Figure 5. It should however be

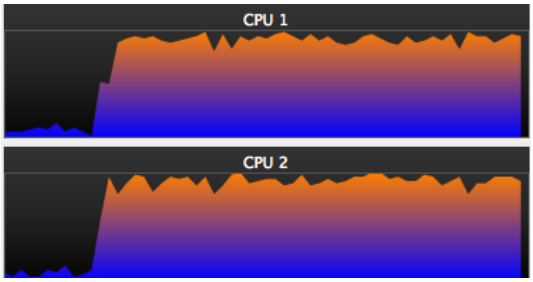

Figure 2: CPU usage with multithreaded version

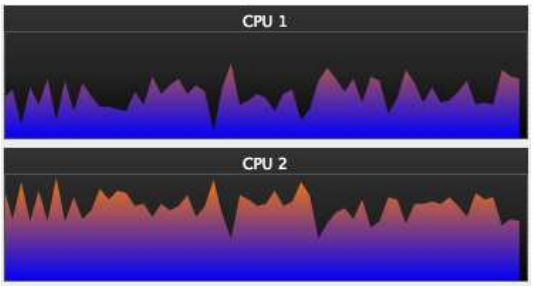

Figure 3: CPU usage with single threaded version

	mono	multi
real	0m51.540s	0m30.786s
user	0m50.992s	0m52.510s
sys	0m0.282s	0m0.599s

Figure 4: Time for 1K particles during 100 instants

	mono	multi
real	3m25.390s	2m4.462s
user	3m24.413s	3m33.905s
sys	0m0.635s	0m1.898s

Figure 5: Time for 2K particles during 100 instants

noticed that the time values presented highly depend on the present FunLoft implementation, which has to be improved in several ways (and particulary, in aspects related to garbage collection). They should only be considered as indications.

4 Related Work

The Threading Building Block[2] (TBB) of Intel is a C++ library which proposes templates to facilitate data-parallel programming. It comes with several examples and benchmarks. No benchmark however concentrates on applications in which threads are strongly synchronised and communicate often, like in the proposal presented here.

Game Of Life, a particular cellular automaton, has been programmed using TBB in [3]. In the multithreaded version, two treads are running, one for computing the next state of cells, and the other to get information from neigbours. Note that the two threads do not communicate at all and just synchronise at each global step of the automaton evolution. Moreover, this architecture does not fit well with machines having more than 2 cores.

Cellular automata have also been considered in [6] in the context of multithreading. By contrast with the computing effort demanded for each particle in the benchmark proposed in this paper, the computing effort needed for each cell of a cellular automaton is usually very light. Cellular automata have been implemented in FunLoft. The array of cells is decomposed in slices of equal length, run by specific native threads. Except for the special case of self-replicating loops (see [6] for details), the benefit of using several threads does not clearly appear in these examples. This, however, is to be confirmed by further experimentations.

The issue of game programming in a multicore context is considered by several papers (for example, in [5]).

Several benchmarks for Haskell are presented in [7]. These benchmarks basically use transactions. It would be interesting to code the collision example using transactions in Haskell, to observe the behavior on multicore architectures.

5 Conclusion

We have proposed a benchmark application for multicore machines. The benchmark illustrates the use of heavily communicating and synchronising threads. A priori, such threads are not good candidates for exploiting the full parallelism offered by multicore machines. However, the benchmark shows that a benefit can still be drawn, because the computing effort that has to be done by each thread is more important than the one needed for synchronisation.

The benchmark has been implemented in FunLoft and the obtained results are given in the paper. Note that, due to the specific characteristics of FunLoft, the code is automatically free of data-races.

References

[1] FunLoft. `http://www.inria.fr/mimosa/rp/FunLoft`.

[2] Intel Threading Building Blocks. `http://osstbb.intel.com/documentation.php`.

[3] Multi Threading Sample - The Game of Life. `http://aeshen.typepad.com/-aeshen/2007/02/the_game_of_life.html`.

[4] Simple Directmedia Layer. `http://www.libsdl.org`.

[5] Game Programming in a Multi Core World, august 2006. Report by Mohamed Eldawy, available at `http://adlcommunity.net`.

[6] Frédéric Boussinot. *Reactive Programming of Cellular Automata.* Inria research report, RR-5183, 2004.

[7] Perfumo C., Sonmez N., Cristal A., Unsal O.S., Valero M., and Harris T. Dissecting Transactional Executions in Haskell. In *Proc. Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

ANNEX: Code of Collision Simulation

This section contains the FunLoft[1] code for the simulation. This code is the one that has been used for producing the results previously given.

It should be noticed that threads in FunLoft (produced as instances of modules using the `thread` construct) are basically user-threads, defined at a logical level. On the opposite, FunLoft schedulers are mapped on physical (native) threads (pthreads in the current implementation). Thus, schedulers are the units executed in real parallelism on multicore machines.

```
/**************************************************
FILE: collision.fl
DESCRIPTION: simulation of collisions
**************************************************/
// number of instants of the simulation
let instants = 100

// number of particles in the simulation
let particle_number = 2000

/**************************************************/
// SCHEDULERS
/**************************************************/
/*
//Monothreaded version: only one scheduler
let sched1 = scheduler
let sched2 = sched1
*/

//Multithreaded version: two synchronised schedulers
let sched1 = scheduler
and sched2 = scheduler

/**************************************************/
// EXTERNAL FUNCTIONS
/**************************************************/
/* External functions returning the applet dimensions, and the size of
   particles. */

let get_maxx : unit -> int
let get_maxy : unit -> int
let get_size : unit -> int

// Interface with the graphical level
let start_graphics : unit -> bool // true = ok
let end_graphics : unit -> unit
let update_display : unit -> unit

// Color type
type color_t =
   BLUE | GREEN | YELLOW | RED |
   CYAN | BLACK | MAGENTA | WHITE

// External function to draw a rectangle
let draw_rectangle :
    int // x
  * int // y
  * int // size in x
  * int // size in y
  * color_t -> unit

/**************************************************/
// SIZE VARIABLES
/**************************************************/
```

```
// Dimension of the applet and size of particles.

let maxx = get_maxx ()-10
let maxy = get_maxy ()-10
let size = get_size ()

/*************************************************/
// PARTICLE
/*************************************************/
/* A particle is a structure holding five references: x and y
coordinates, x and y speeds, and color. */

type particle_t = Particle of
      int ref      * // x coord
      int ref      * // y coord
      int ref      * // x speed
      int ref      * // y speed
      color_t ref    // color

/*************************************************/
// Access to particle coordinates
let x_coord (s) =
   match s with Particle (x,_,_,_,_) -> !x end

let y_coord (s) =
   match s with Particle (_,y,_,_,_) -> !y end

/*************************************************/
// Maximum speed of particles
let max_speed = 5

// Random speed (positive or negative, not zero)
let random_speed (m) =
  let x = random_int (2) in
  let v = random_int (m) + 1 in
     if x = 0 then v else -v

/* Creation of a new particle. The particle is randomly placed, and
   has a random speed */

let new_particle (color) =
   let x = random_int (maxx) in
   let y = random_int (maxy) in
   let sx = random_speed (max_speed) in
   let sy = random_speed (max_speed) in
      Particle (ref x,ref y,ref sx,ref sy,ref color)

/*************************************************/
// Invert the speed of a particle
let invert_x_speed (s) =
   match s with Particle (_,_,sx,_,_) -> sx := -!sx end

let invert_y_speed (s) =
   match s with Particle (_,_,_,sy,_) -> sy := -!sy end

/*************************************************/
// Moves a particle in the four directions
let go_right (s,dist) =
   match s with Particle (x,_,_,_,_) -> x := !x+dist end

let go_down (s,dist) =
   match s with Particle (_,y,_,_,_) -> y := !y+dist end

let go_left (s,dist) = go_right (s,-dist)

let go_up (s,dist) = go_down (s,-dist)

/*************************************************/
```

```
// DRAWING PROCESSOR
/*************************************************/
/* An auxiliary type is defined to hold images of particles. */

type image_t = Image of
    int // x
  * int // y
  * color_t

// Drawing function: a particle is drawn as a square
let draw_image (i) =
   match i with Image (x,y,c) -> draw_rectangle (x,y,size,size,c) end

// The event used for drawing orders
let draw_event = event

/* The module that process drawing orders: at each instant, values of
   draw_event are collected and the function draw_image is called for
   each of them */

let module draw_processor () =
   loop for_all_values draw_event with a -> draw_image (a)

/*************************************************/
// DRAWING BEHAVIOR
/*************************************************/
// Generate a drawing order for the particle at each instant.
let module draw_behavior (me) =
 loop
   begin
      match me with Particle (x,y,_,_,c) ->
         generate draw_event with Image (!x,!y,!c)
      end;
      cooperate
   end

/*************************************************/
// INERTIAL BEHAVIOR
/*************************************************/
/* Give inertia to the particle at each instant. Inertia simply
   increment coordinates by speed. */

let module inertia_behavior (me) =
 loop
   begin
      match me with Particle (x,y,sx,sy,_) ->
         begin x:=!x+!sx; y:=!y+!sy end
      end;
      cooperate
   end

/*************************************************/
// BOUNCING BEHAVIOR
/*************************************************/
// Let the particle bounce on the applet borders at each instant.
let module bounce_behavior (me) =
  loop
    begin
      match me with Particle (x,y,sx,sy,_) ->
        begin
          if !x < 0 then
            begin invert_x_speed (me); x:=-!x end
          else if !x > maxx then
            begin invert_x_speed (me); x:=maxx-(!x-maxx) end
          else ();
          if !y < 0 then
            begin invert_y_speed (me); y:=-!y end
          else if !y > maxy then
```

```
            begin invert_y_speed (me); y:=maxy-(!y-maxy) end
          else ();
        end
      end;
      cooperate
    end

/*****************************************************/
// COLLIDE BEHAVIOR
/*****************************************************/
// Square of the maximum distance for collision processing.
let max_collision = 200

// Auxiliary type to hold particle coordinates.
type coord_t = Coord of int * int

/* The collision function processes possible collision beetwen a
   particle and another one given by its coordinates.  First, the
   distance between the two particles is computed, then it is checked
   (it should be strictly positive and less than a maximal value). If
   the check is positive, then speed of the particle is inverted and
   the particle is slightly moved. */

let collision (me,other) =
  match other with Coord (x,y) ->
   let div = 3 in
   let dx = x - x_coord (me) in
   let dy = y - y_coord (me) in
   let ds = (dx*dx) + (dy*dy) in
     if (ds > 0) && (ds < max_collision) then
       begin
         invert_x_speed (me);
         invert_y_speed (me);
         if dx > 0 then go_left (me,dx/div) else go_right (me,-dx/div);
         if dy > 0 then go_up (me,dy/div) else go_down (me,-dy/div)
       end
     else ()
  end

/* At each instant, the collide behavior generates the collide event
with the particle coordinates, it collects all the coords sent by all
other particles, and it process collision with them. This algorithm is
not optimal ((particle_number-1)^2; it could be divided by 2). */

let module collide_behavior (me,collide_event) =
 loop
   begin
     generate collide_event with Coord (x_coord (me),y_coord (me));
     for_all_values collide_event with other -> collision (me,other)
   end

/*****************************************************/
// PARTICLE BEHAVIOR
/*****************************************************/
/* Each particle is animated by four threads: one for inertia,
   one for bouncing on the applet borders, one for collision
   processing, and one for graphics. All these threads share the
   particle. */

let module particle_behavior  (collide_event,color) =
  let s = new_particle (color) in
    begin
      thread inertia_behavior (s);
      thread bounce_behavior (s);
      thread collide_behavior (s,collide_event);
      thread draw_behavior (s);
    end
```

```
/*********************************************/
// GRAPHICS
/*********************************************/
/* Initialisation of graphical environment. */

let initialise_graphics () =
    if start_graphics () then
        begin
           print_string ("\ndisplay "); print_int (maxx);
           print_string ("x"); print_int (maxy);
           print_string (" ok\n");
        end
    else
       begin
         print_string ("can't initialize display\n");
         quit (1)
       end

/* At each instant, display is updated and a white background is
   issued. */

let module graphics () =
  begin
     initialise_graphics ();
     loop
        begin
          update_display ();
          draw_rectangle (0,0,maxx+20,maxy+20,WHITE);
          cooperate;
        end
  end

/*********************************************/
// SYSTEM BUILD
/*********************************************/
/* Half of the particles are launched in sched1 and are painted in a
   first color, and the other half are launched in sched2 and are
   painted in another color. They all share the same collide event. */

let module system () =
   let collide_event = event in
   let half = particle_number / 2 in
   begin
     link sched1 do
        repeat half  do
          thread particle_behavior (collide_event,CYAN);
     link sched2 do
        repeat half do
          thread particle_behavior (collide_event,BLACK);
   end

/*********************************************/
// TRACE INSTANTS
/*********************************************/
/* Tracing of instants and termination of simulation. Tracing should
   be launched in a scheduler in which objects are present (not in the
   implicit scheduler). */

let module trace_instants () =
 let x = ref 0 in
  begin
    repeat instants do
    begin
      print_int (!x); print_string (" "); flush ();
      x++;
      cooperate;
    end;
    print_string ("\nend of simulation\n");
```

```
    end_graphics ();
    quit (0)
  end

/*********************************************/
// MAIN
/*********************************************/
/* The main module launches four threads: one for graphics, one to
   collect drawing demands, one to trace instants, and one for
   building the system. They are all arbitrarily launched in sched1
   (sched2 would be ok as well). */

let module main () =
  link sched1 do
    begin
      thread graphics ();
      thread draw_processor ();
      thread trace_instants ();
      thread system ();
    end

/*********************************************/
```